

# Objects and Local State

Mattox Beckman  
beckman@iit.edu

Illinois Institute of Technology

- We will use the following functions during our discussion....

```
1 let pi1 (x,y) = x
2 let pi2 (x,y) = y
3 let report (x,y) = print_string "Point: ";
4                     print_int x;
5                     print_string ",";
6                     print_int y;
7                     print_newline ()
8 let movept (x,y) (dx,dy) = (x+dx,y+dy)
```

In this lecture we extend the idea of local state from last time to create a simple implementation of objects, and discuss its limitations. We will also show the message dispatch model of objects, which allows for inheritance and virtual functions. Your objectives:

- Be able to explain what an object is.
- Know how to implement an object using records and HOFs.
- Know how to implement an object using a message dispatcher.
- Be able compare the record and dispatcher models.
- Major goal 1: be able to simulate objects in a language lacking them.
- Major goal 2: understand how objects work “under the hood”.

Here is an example of a point using local state.

```
1 let mkPoint myloc =
2   let myloc = ref myloc in
3   ( myloc,
4     (fun () -> pi1 !myloc),
5     (fun () -> pi2 !myloc),
6     (fun () -> report !myloc),
7     (fun dl -> myloc := movept !myloc dl) )
```

- This defines a tuple of functions that share a common state.
- It is cumbersome to use.

```
let (lref,getX,getY,show,move) = mkPoint (2,4);;
```

```

1 type point = {
2   loc : (int * int) ref; getx : unit -> int;
3   gety : unit -> int; draw : unit -> unit;
4   move : int * int -> unit;
5 }
6 let mkrPoint newloc =
7   let myloc = ref newloc in
8   { loc = myloc;
9     getx = (fun () -> pi1 !myloc);
10    gety = (fun () -> pi2 !myloc);
11    draw = (fun () -> report !myloc);
12    move = (fun dl -> myloc := movept !myloc dl)}

```

By the way, this lecture is really about recursion.

```

1 let mkPoint newloc =
2   let rec this =
3     { loc = ref newloc;
4       getx = (fun () -> pi1 !(this.loc));
5       gety = (fun () -> pi2 !(this.loc));
6       draw = (fun () -> report !(this.loc));
7       move = (fun dl ->
8                 this.loc := movept !(this.loc) dl)}
9   in this;;

```

We can store “this” explicitly in the record if we want.

What is an object?

- Data and functions are grouped together.
- Functions have their own local state.
- Objects can send and receive *messages*.
- Objects can refer to *themselves*.

This has a profound effect on the way programs are written.

Remember the basic premise of this course: how you think about data has a great impact on the way a program is written.

- How is the mkrPoint example like an object?
- How is the mkrPoint example not like an object?

- The record point contains references to the fields. If you copy a point, the data does **not** get copied!

```

1 # let p1 = mkPoint (4,7);;
2 val p1 : point = {loc={contents=4, 7}; ...}
3 # let p2 = mkPoint (6,2);;
4 val p2 : point = {loc={contents=6, 2}; ...}
5 # let p3 = p1;;
6 val p3 : point = {loc={contents=4, 7}; ...}
7 # p1.move (5,5);;
8 - : unit = ()
9 # p3;;
10 - : point = {loc={contents=9, 12}; ...}

```

We used a record to implement a type for points.

Advantages:

- Every method had its own name and type.
- Simple syntax for manipulating the object.
- It's fast: we know at compile time which method is being called.

Disadvantages:

- Inheritance is very difficult with this model.
- Adding a new message type means updating *everything*.

Last time we said that an object is a kind of data that can *receives messages* from the program or other objects.

- Q: How do we normally represent messages?
- A: With strings!

Let a point object be a function which takes a string and returns an appropriate function matching that string.

- Question: Suppose *p* is our point object. What will be its type?

```

1 let mkPoint x y =
2   let x = ref x in
3   let y = ref y in
4   fun st ->
5     match st with
6     | "getx" -> (fun _ -> !x)
7     | "gety" -> (fun _ -> !y)
8     | "movx" -> (fun nx -> x := !x + nx; nx)
9     | "movy" -> (fun ny -> y := !y + ny; ny)
10    | _ -> raise (Failure "Unknown message.")

```

All methods now have to have type `int -> int`.

- Warmup exercise: How would we add a `report` method?
- Another one: How would we add `this` support?

Let's say we want a `fastpoint`, which moves twice as fast as the original point. What does it mean for `fastpoint` to be a *subclass* of `point`?

- `fastpoint` should respond to the same messages.
  - It may override some of them.
  - It may add its own.
  - It may **not** remove any methods.
- The `fastpoint` object will need access to some of the data in `point`.

- Two entities involved: the superclass (`point`) and the subclass (`fastpoint`).
- `fastpoint` needs to create an instance of `point`.
- `point` construction needs to return the “public” data to `fastpoint`.
- `fastpoint` returns a dispatcher:
  - if the `fastpoint` dispatcher can handle a message, it does.
  - Otherwise, it *sends the message* to `point`.

```

1 let mkSuperPoint x y =
2   let x = ref x in
3   let y = ref y in
4   ((x,y), (* This part returns the local state *))
5   fun st ->
6     match st with
7     | "getx" -> (fun _ -> !x)
8     | "gety" -> (fun _ -> !y)
9     | "movx" -> (fun nx -> x := !x + nx; nx)
10    | "movy" -> (fun ny -> y := !y + ny; ny)
11    | _ -> raise (Failure "Unknown message.");//
12 val mkSuperPoint : int -> int ->
13   (int ref * int ref) * (string -> int -> int) = <fu

```

```

1 let mkFastpoint x y =
2   let ((x,y),super) = mkSuperPoint x y in
3   fun st ->
4     match st with
5     | "movx" -> (fun nx -> x := !x + 2 * nx; nx)
6     | "movy" -> (fun ny -> y := !y + 2 * ny; ny)
7     | _ -> super st;;

```

- This technique is flexible; we can add methods very easily.
- But it's also slow. Imagine if we had a chain of 20 classes....

- Methods and variables are kept in a table: a fixed location.
- “this” is an implicit argument, allowing only one copy of the function to be needed.
- Virtual methods are kept in a *vtable*, which counts as local data.

Local data for `point` or `fastpoint`:

x	value of x
y	value of y
vtable	pointer to vtable

Vtable for `point`:

movx	pointer to <code>point.movx</code>
movy	pointer to <code>point.movy</code>

(`fastpoint` vtable is similar.) `getx`, etc. is static.

- Other languages (i.e., smalltalk) use a technique very similar to this one.
- Java uses the “every object is of type `Object`” technique.
- A strong type system makes it somewhat cumbersome to simulate objects. You either have to:
  - define a new type to encompass all objects, or
  - force all methods to have the same type.
- Important concept: *polymorphism* — when functions can operate on multiple types. (This is different than *overloading* — when multiple functions exist with the same name, but different inputs.)

- Objects have a lot of flexibility, and allow us to create useful abstractions.
- They can be implemented using functions.
- These are useful enough in practice, and difficult enough to implement, that most modern languages now include them, including OCaml. (That’s where the O comes from.)

```

1 # let p1,p2,p3,p4 = mkPoint 2 3, mkPoint 3 2,
2   mkFastpoint 5 3, mkFastpoint 3 9;;
3 # List.map (fun pt -> pt "report" 0)
4   [p1; p2; p3; p4];;
5 Point: 2,3   points
6 Point: 3,2
7 Point: 5,3   fastpoints
8 Point: 3,9

```

The function passed to `map` will use both `point` and `fastpoint` types.

You have seen polymorphism in the course before.